chapter

14

Functions

14.1 Introduction

Functions are subprograms, and subprograms are the soul of modern programming languages. Functions provide the programmer with a way to enlarge the set of elementary building blocks with which to write programs. That is, they enable the programmer to extend the set of operations and constructs natively supported by the language to include new primitives. Functions are such an important concept that they have been part of languages since the very early days, and support for them is provided directly in all instruction set architectures, including the LC-3.

Why are they so important? Functions (or procedures, or subroutines, or methods—all of which are variations of the same theme) enable *abstraction*. That is, they increase our ability to separate the "function" of a component from the details of how it accomplishes that "function." Once the component is created and we understand its construction, we can use the component as a building block without giving much thought to its detailed implementation. Without abstraction, our ability to create complex systems such as computers, and the software that runs on them, would be seriously impaired.

Functions are not new to us. We have have been using variants of functions ever since we programmed subroutines in LC-3 assembly language; while there are syntactic differences between subroutines in LC-3 assembly and functions in C, the concepts behind them are largely the same.

The C programming language is heavily oriented around functions. A C program is essentially a collection of functions. Every statement belongs to one (and only one) function. All C programs start and finish execution in the function `main`.

The function `main` might call other functions along the way, and they might, in turn, call more functions. Control eventually returns to the function `main`, and when `main` ends, the program ends (provided something did not cause the program to terminate prematurely).

In this chapter, we provide an introduction to functions in C. We begin by examining several short programs in order to get a sense of the C syntax involving functions. Next, we examine how functions are implemented, examining the low-level operations necessary for functions to work in high-level languages. In the last part of the chapter, we apply our problem-solving methodology to some programming problems that benefit from the use of functions.

14.2 Functions in C

Let's start off with a simple example of a C program involving functions. Figure 14.1 is a program that prints a message using a function named `PrintBanner`. This program begins execution at the function `main`, which then calls the function `PrintBanner`. This function prints a line of text consisting of the = character to the output device.

`PrintBanner` is the simplest form of a function: it requires no input from its caller to do its job, and it provides its caller with no output data (not counting the banner printed to the screen). In other words, no arguments are passed from `main` to `PrintBanner` and no value is returned from `PrintBanner` to `main`. We refer to the function `main` as the *caller* and to `PrintBanner` as the *callee*.

14.2.1 A Function with a Parameter

The fact that `PrintBanner` and `main` require no exchange of information simplifies their interface. In general, however, we'd like to be able to pass some information between the caller and the callee. The next example demonstrates

```
1  #include <stdio.h>
2
3  void PrintBanner();       /* Function declaration */
4
5  int main()
6  {
7     PrintBanner();         /* Function call        */
8     printf("A simple C program.\n");
9     PrintBanner();
10 }
11
12 void PrintBanner()       /* Function definition  */
13 {
14    printf("============================\n");
15 }
```

Figure 14.1 A C program that uses a function to print a banner message

```
1  #include <stdio.h>
2
3  int Factorial(int n);                    /*! Function Declaration !*/
4
5  int main()                               /*  Definition for main  */
6  {
7    int number;                            /*  Number from user     */
8    int answer;                            /*  Answer of factorial  */
9
10   printf("Input a number: ");            /*  Call to printf       */
11
12   scanf("%d", &number);                  /*  Call to scanf        */
13
14   answer = Factorial(number);            /*! Call to factorial    !*/
15
16   printf("The factorial of %d is %d\n", number, answer);
17 }
18
19 int Factorial(int n)                     /*! Function Definition  !*/
20 {
21   int i;                                 /*  Iteration count      */
22   int result = 1;                        /*  Initialized result   */
23
24   for (i = 1; i <= n; i++)               /*  Calculate factorial  */
25     result = result * i;
26
27   return result;                         /*! Return to caller     !*/
28 }
```

Figure 14.2 A C program to calculate factorial

how this is done in C. The code in Figure 14.2 contains a function `Factorial` that performs an operation based on an input parameter.

`Factorial` performs a multiplication of all integers between 1 and `n`, where `n` is the value provided by the caller function (in this case `main`). The calculation performed by this function can be algebraically stated as:

$$\text{factorial}(n) = n! = 1 \times 2 \times 3 \times \ldots \times n$$

The value calculated by this function is named `result` in the C code in Figure 14.2. Its value is returned (using the `return` statement) to the caller. We say that the function `Factorial` requires a single integer *argument* from its caller, and it *returns* an integer value back to its caller. In this particular example, the variable `answer` in the caller is assigned the return value from `Factorial` (line 14).

Let's take a closer look at the syntax involved with functions in C. In the code in Figure 14.2, there are four lines that are of particular interest to us. The *declaration* for `Factorial` is at line 3. Its *definition* starts at line 19. The call to `Factorial` is at line 14; this statement invokes the function. The return from `Factorial` back to its caller is at line 27.

The Declaration

In the preceding example, the function declaration for `Factorial` appears at line 3. What is the purpose of a function's declaration? It informs the compiler about some relevant properties of the function in the same way a variable's declaration informs the compiler about a variable. Sometimes called a *function prototype*, a function declaration contains the name of the function, the type of value it returns, and a list of input values it expects. The function declaration ends with a semicolon.

The first item appearing in a function's declaration is the type of the value the function returns. The type can be any C data type (e.g., `int`, `char`, `double`). This type describes the type of the single output value that the function produces. Not all functions return values. For example, the function `PrintBanner` from the previous example did not return a value. If a function does not return a value, then its return type must be declared as `void`, indicating to the compiler that the function returns nothing.

The next item on the declaration is the function's name. A function's name can be any legal C identifier. Often, programmers choose function names somewhat carefully to reflect the actions performed by the function. `Factorial`, for example, is a good choice for the function in our example because the mathematical term for the operation it performs is *factorial*. Also, it is good style to use a naming convention where the names of functions and the names of variables are easily distinguishable. In the examples in this book, we do this by capitalizing the first character of all function names, such as `Factorial`.

Finally, a function's declaration also describes the type and order of the input *parameters* required by the function. These are the types of values that the function expects to receive from its callers and the order in which it expects to receive them. We can optionally specify (and often do) the name of each parameter in the declaration. For example, the function `Factorial` takes one integer value as an input parameter, and it refers to this value internally as `n`. Some functions may not require any input. The function `PrintBanner` requires no input parameters; therefore its parameter list is empty.

The Call

Line 14 in our example is the function call that invokes `Factorial`. In this statement, the function `main` calls `Factorial`. Before `Factorial` can start, however, `main` must transmit a single integer value to `Factorial`. Such values within the caller that are transmitted to the callee are called *arguments*. Arguments can be any legal expression, but they should match the type expected by the callee. These arguments are enclosed in parentheses immediately after the callee's name. In this example, the function `main` passes the value of the variable `number` as the argument. The value returned by `Factorial` is then assigned to the integer variable `answer`.

The Definition

The code beginning at line 19 is the function definition for `Factorial`. Notice that the first line of the definition matches the function declaration (however, minus the

semicolon). Within the parentheses after the name of the function is the function's *formal parameter list*. The formal parameter list is a list of variable declarations, where each variable will be initialized with the corresponding argument provided by the caller. In this example, when `Factorial` is called on line 14, the parameter `n` will be initialized to the value of `number` from `main`. From every place in the program where a function is called, the actual arguments appearing in each call should match the type and ordering of the formal parameter list.

The function's body appears in the braces following the parameter list. A function's body consists of declarations and statements that define the computation the function performs. Any variable declared within these braces is local to the function.

A very important concept to realize about functions in C is that none of the local variables of the caller are explicitly visible by the callee function. And in particular, `Factorial` cannot modify the variable `number`. In C, the arguments of the caller are *passed as values* to the callee.

The Return Value

In line 27, control passes back from `Factorial` to the caller `main`. Since `Factorial` is returning a value, an expression must follow the `return` keyword, and the type of this expression should match the return type declared for the function. In the case of `Factorial`, the statement `return result;` transmits the calculated value stored in `result` back to the caller. In general, functions that return a value must include at least one `return` statement in their body. Functions that do not return a value—functions declared as type `void`—do not require a `return` statement; the `return` is optional. For these functions, control passes back to the caller after the last statement has executed.

What about the function `main`? Its type is `int` (as required by the ANSI standard), yet it does not contain a return. Strictly speaking, we should include a `return 0` at the end of `main` in the examples we've seen thus far. In C, if a non-`void` function does not explicitly return a value, the value of the last statement is returned to the caller. Since `main`'s return value will be ignored by most callers (who are the callers of `main`?), we've omitted them in the text to make our examples more compact.

Let's summarize these various syntactic components: A function declaration (or prototype) informs the compiler about the function, indicating its name, the number and types of parameters the function expects from a caller, and the type of value the function returns. A function definition is the actual source code for the function. The definition includes a formal parameter list, which indicates the names of the function's parameters and the order in which they will be expected from the caller. A function is invoked via a function call. Input values, or arguments, for the function are listed within the parentheses of the function call. Literally, the value of each argument listed in the function call is assigned to the corresponding parameter in the parameter list, the first argument assigned to the first parameter, the second argument to the second parameter, and so forth. The return value is the output of the function, and it is passed back to the caller function.

14.2.2 Example: Area of a Ring

We further demonstrate C function syntax with a short example in Figure 14.3. This C program calculates the area of a circle that has a smaller circle removed from it. In other words, it calculates the area of a ring with a specified outer and inner radius. In this program, a function is used to calculate the area of a circle with a given radius. The function `AreaOfCircle` takes a single parameter of type `double` and returns a `double` value back to the caller.

The following point is important for us to reiterate: when function `AreaOfCircle` is active, it can "see" and modify its local variable `pi` and its parameter `radius`. It cannot, however, modify any of the variables within the function `main`, except via the value it returns.

The function `AreaOfCircle` in this example has a slightly different usage than the functions that we've seen in the previous examples in this chapter. Notice that there are multiple calls to `AreaOfCircle` from the function `main`. In this case, `AreaOfCircle` performs a useful, primitive computation such that encapsulating it into a function is beneficial. On a larger scale, real programs will include functions that are called from hundreds or thousands of different places. By forming

```
1  #include <stdio.h>
2
3  /* Function declarations */
4  double AreaOfCircle(double radius);
5
6  int main()
7  {
8    double outer;                          /* Outer radius */
9    double inner;                          /* Inner radius */
10   double areaOfRing;                     /* Area of ring */
11
12   printf("Enter inner radius: ");
13   scanf("%lf", &outer);
14
15   printf("Enter outer radius: ");
16   scanf("%lf", &inner);
17
18   areaOfRing = AreaOfCircle(outer) - AreaOfCircle(inner);
19   printf("The area of the ring is %f\n", areaOfRing);
20 }
21
22 /* Calculate area of circle given a radius */
23 double AreaOfCircle(double radius)
24 {
25   double pi = 3.14159265;
26
27   return pi * radius * radius;
28 }
```

Figure 14.3 A C program calculates the area of a ring

`AreaOfCircle` and similar primitive operations into functions, we potentially save on the amount of code in the program, which is beneficial for code maintenance. The program also takes on a better structure. With `AreaOfCircle`, the intent of the code is more visibly apparent than if the formula were directly embedded in-line.

Some of you might remember our discussion on constant values from Section 12.6.2, where we argue that the variable `pi` should be declared as a constant using the `const` qualifier on line 25 of the code. We omit it here to make the example accessible to those who that might have skipped over the Additional Topics section of Chapter 12.

14.3 Implementing Functions in C

Let's now take a closer look at how functions in C are implemented at the machine level. Functions are the C equivalent of subroutines in LC-3 assembly language (which we discussed in Chapter 9), and the core of their operation is the same. In C, making a function call involves three basic steps: (1) the parameters from the caller are passed to the callee and control is transfered to the callee, (2) the callee does its task, (3) a return value is passed back to the caller, and control returns to the caller. An important constraint that we will put on the calling mechanism is that a function should be *caller-independent*. That is, a function should be callable from any function. In this section we will examine how this is accomplished using the LC-3 to demonstrate.

14.3.1 Run-Time Stack

Before we proceed, we first need to discuss a very important component of functions in C and other modern programming languages. We require a way to "activate" a function when it is called. That is, when a function starts executing, its local variables must be given locations in memory. Let us explain:

Each function has a memory template in which its local variables are stored. Recall from our discussion in Section 12.5.2 that an activation record for a function is a template of the relative positions of its local variables in memory. Each local variable declared in a function will have a position in the activation record. Recall that the frame pointer (R5) indicates the start of the activation record. *Question:* Where in memory does the activation record of a function reside? Let's consider some options.

Option 1: The compiler could systematically assign spots in memory for each function to place its activation record. Function A might be assigned memory location X to place its activation record, function B might be assigned location Y, and so forth, provided, of course, that the activation records do not overlap. While this seems like the most straightforward way to manage the allocation, a serious limitation arises with this option. What happens if function A calls itself? We call this *recursion*, and it is a very important programming concept that we will discuss in Chapter 17. If function A calls itself, then the callee version of function A will overwrite the local values of the caller version of function A, and

the program will not behave as we expect it to. For the C programming language, which allows recursive functions, option 1 will not work.

Option 2: Every time a function is called, an activation record is allocated for it in memory. And when the function returns to the caller, its activation record is reclaimed to be assigned later to another function. While this option appears to be conceptually more difficult than option 1, it permits functions to be recursive. Each *invocation* of a function gets its own space in memory for its locals. For example, if function A calls function A, the callee version will be allocated its own activation record for storing local values, and this record will be different than the caller's. There is a factor that reduces the complexity of making option 2 work: The calling pattern of functions (i.e., function A calls B which calls C, etc.) can be easily tracked with a stack data structure (Chapter 10). Let us demonstrate with an example.

The code in Figure 14.4 contains three functions, `main`, `Watt`, and `Volta`. What each function does is not important for this example, so we've omitted some of their details but provided enough so that the calling pattern between them is

```
1   int main()
2   {
3     int a;
4     int b;
5
6     :
7     b = Watt(a);            /* main calls both      */
8     b = Volta(a, b);
9   }
10
11  int Watt(int a)
12  {
13    int w;
14
15    :
16    w = Volta(w, 10);       /* Watt calls Volta     */
17
18    return w;
19  }
20
21  int Volta(int q, int r)
22  {
23    int k;
24    int m;
25
26    :                       /* Volta calls no one */
27    return k;
28  }
```

Figure 14.4 Code example that demonstrates the stack-like nature of function calls

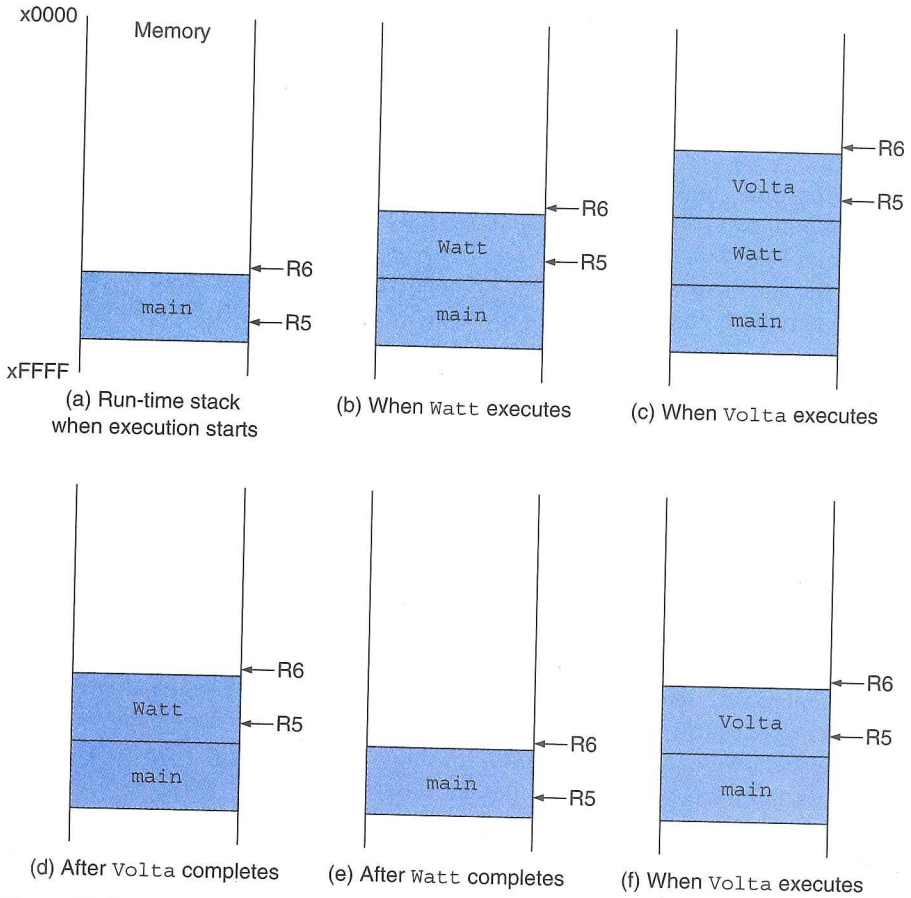

Figure 14.5 Several snapshots of the run-time stack while the program outlined in Figure 14.4 executes

apparent. The function `main` calls `Watt` and `Watt` calls `Volta`. Eventually, control returns back to `main` which then calls `Volta`.

Each function has an activation record that consists of its local variables, some bookkeeping information, and the incoming parameters from the caller (we'll mention more about the parameters and bookkeeping information in the subsequent paragraphs). Whenever a function is called, its activation record will be allocated somewhere in memory, and as we indicated in the previous paragraph, in a stack-like fashion. This is illustrated in the diagrams of Figure 14.5.

Each of the shaded regions represents the activation record of a particular function call. The sequence of figures shows how the run-time stack grows and shrinks as the various functions are called and return to their caller. Keep in mind that, as we push items onto the stack, the top of the stack moves, or "grows," toward lower-numbered memory locations.

Figure 14.5(a) is a picture of the run-time stack when the program starts execution. Since the execution of a C program starts in `main`, the activation record

for `main` is the first to be allocated on the stack. Figure 14.5(b) shows the run-time stack immediately after `Watt` is called by `main`. Notice that the activation records are allocated in a stack-like fashion. That is, whenever a function is called, its activation record is *pushed* onto the stack. Whenever the function returns, its activation is *popped* off the stack. Figure 14.5 parts (c) through (f) show the state of the run-time stack at various points during the execution of this code. Notice that R5 points to some internal location within the activation record (it points to the base of the local variables). Also notice how R6 always points to the very top of the stack—it is called the stack pointer. Both of these registers have a key role to play in the implementation of the run-time stack and of functions in C in general.

14.3.2 Getting It All to Work

It is clear that there is a lot of work going on at the machine level when a function is called. Parameters must be passed, activation records pushed and popped, control moved from one function to another. Some of this work is accomplished by the caller, some by the callee.

To accomplish all of this, the following steps are required: First, code in the caller function copies its arguments into a region of memory accessible by the callee. Second, the code at the beginning of the callee function pushes its activation record onto the stack and saves some bookkeeping information so that when control returns to the caller, it appears to the caller as if its local variables and registers were untouched. Third, the callee does its thing. Fourth, when the callee function has completed its job, its activation record is popped off the run-time stack and control is returned to the caller. Finally, once control is back in the caller, code is executed to retrieve the callee's return value.

Now we'll examine the actual LC-3 code for carrying out these operations. We do so by examining the LC-3 code associated with the following function call: `w = Volta(w, 10);` from line 18 of the code in Figure 14.4.

The Call

In the statement `w = Volta(w, 10);`, the function `Volta` is called with two arguments. The value returned by `Volta` is then assigned to the local integer variable `w`. In translating this function call, the compiler generates LC-3 code that does the following:

1. Transmits the value of the two arguments to the function `Volta` by pushing them directly onto the top of the run-time stack. Recall that R6 points to the top of the run-time stack. That is, it contains the address of the data item currently at the top of the run-time stack. To push an item onto the stack, we first decrement R6 and then store the data value using R6 as a base address. In the LC-3, the arguments of a C function call are pushed onto the stack from **right-to-left** in order they appear in the function call. In the case of `Watt`, we will first push the value 10 (rightmost argument) and then the value of `w`.
2. Transfers control to `Volta` via the `JSR` instruction.

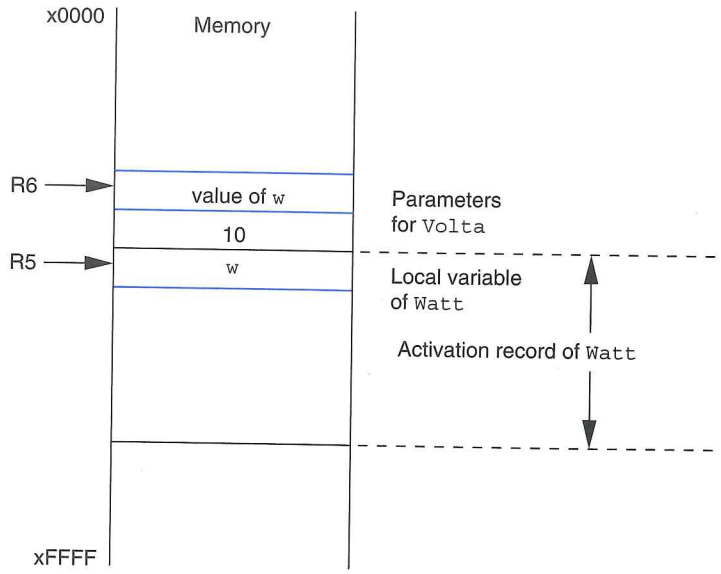

Figure 14.6 The run-time stack Watt pushes the values it wants to pass to Volta

The LC-3 code to perform this function call looks like this:

```
AND  R0, R0, #0   ; R0 <- 0
ADD  R0, R0, #10  ; R0 <- 10
ADD  R6, R6, #-1  ;
STR  R0, R6, #0   ; Push 10

LDR  R0, R5, #0   ; Load w
ADD  R6, R6, #-1  ;
STR  R0, R6, #0   ; Push w

JSR  Volta
```

Figure 14.6 illustrates the modifications made to the run-time stack by these instructions. Notice that the argument values are pushed immediately on top of the activation record of the caller (Watt). The activation record for the callee (Volta) will be constructed on the stack directly on top of the record of the caller.

Starting the Callee Function

The instruction executed immediately after the JSR in the function Watt is the first instruction in the callee function Volta.

The code at the beginning of the callee handles some important bookkeeping associated with the call. The very first thing is the allocation of memory for the return value. The callee will push a memory location onto the stack by decrementing the stack pointer. And this location will be written with the return value prior to the return to the caller.

Next, the callee function saves enough information about the caller so that eventually when the called has finished, the caller can correctly regain program control. In particular, we will need to save the caller's return address, which is in R7 (Why is it in R7? Recall how the JSR instruction works.) and the caller's frame pointer, which is in R5. It is important to make a copy of the caller's frame pointer, which we call the *dynamic link*, so that when control returns to the caller it will be able once again to access its local variables. If either the return address or the dynamic link is destroyed, then we will have trouble restarting the caller correctly when the callee finishes. Therefore it is important that we make copies of both in memory.

Finally, when all of this is done, the callee will allocate enough space on the stack for its local variables by adjusting R6, and it will set R5 to point to the base of its locals.

To recap, here is the list of actions that need to happen at the beginning of a function:

1. The callee saves space on the stack for the return value. The return value is located immediately on top of the parameters for the callee.
2. The callee pushes a copy of the return address in R7 onto the stack.
3. The callee pushes a copy of the dynamic link (caller's frame pointer) in R5 onto the stack.
4. The callee allocates enough space on the stack for its local variables and adjusts R5 to point to the base of the local variables and R6 to point to the top of the stack.

The code to accomplish this for `Volta` is:

```
Volta:
  ADD R6, R6, #-1   ; Allocate spot for the return value

  ADD R6, R6, #-1   ;
  STR R7, R6, #0    ; Push R7 (Return address)

  ADD R6, R6, #-1   ; Push R5 (Caller's frame pointer)
  STR R5, R6, #0    ; We call this the dynamic link

  ADD R5, R6, #-1   ; Set new frame pointer
  ADD R6, R6, #-2   ; Allocate memory for Volta's locals
```

Figure 14.7 summarizes the changes to memory accomplished by the code we have encountered so far. The layout in memory of these two activation records—one for `Watt` and one for `Volta`—is apparent. Notice that some entries of the activation record of `Volta` are written by `Watt`. In particular, these are the parameter fields of `Volta`'s activation record. `Watt` writes the value of its local variable `w` as the first parameter and the value 10 for the second parameter. Keep in mind that these values are pushed from right to left according to their position in the function call. Therefore, the value of `w` appears on top of the value 10. Once

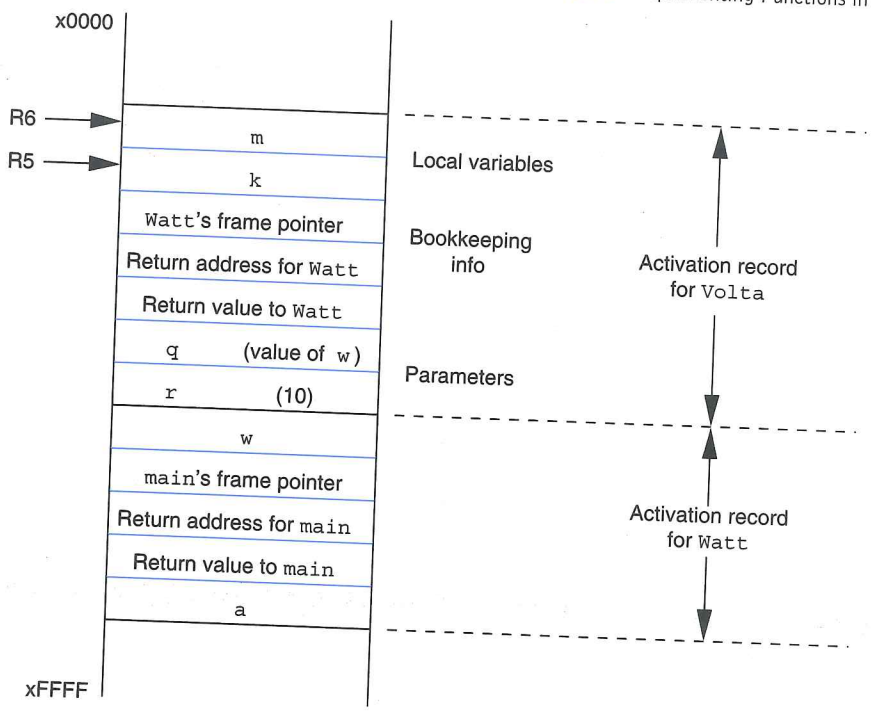

Figure 14.7 The run-time stack after the activation record for `Volta` is pushed onto the stack

invoked, `Volta` will refer to these values with the names `q` and `r`. Question: What are the initial values of `Volta`'s local variable? Recall from Chapter 11 that local variables such as these are uninitialized. See Exercise 14.10 for an exercise on the initial values of local variables.

Notice that each activation record on the stack has the same structure. Each activation record contains locations for the function's local variables, for the bookkeeping information (consisting of the caller's return address and dynamic link), the return value, and the function's parameters.

Ending the Callee Function

Once the callee function has completed its work, it must perform several tasks prior to returning control to the caller function. Firstly, a function that returns a value needs a mechanism for the return value to be transmitted properly to the caller function. Secondly, the callee must pop the current activation record. To enumerate,

1. If there is a return value, it is written into the return value entry of the activation record.
2. The local variables are popped off the stack.
3. The dynamic link is restored.
4. The return address is restored.
5. The `RET` instruction returns control to the caller function.

The LC-3 instructions corresponding to this for `Volta` are

```
LDR  R0, R5, #0   ; Load local variable k
STR  R0, R5, #3   ; Write it in return value slot

ADD  R6, R5, #1   ; Pop local variables

LDR  R5, R6, #0   ; Pop the dynamic link
ADD  R6, R6, #1   ;

LDR  R7, R6, #0   ; Pop the return address
ADD  R6, R6, #1   ;

RET
```

The first two instructions write the return value, which in this case is the local variable `k`, into the return value entry of `Volta`'s activation record. Next, the local variables are popped by moving the stack pointer to the location immediately below the frame pointer. The dynamic link is restored, then the return address is restored, and finally we return to the caller.

You should keep in mind that even though the activation record for `Volta` is popped off the stack, the values remain in memory.

Returning to the Caller Function

After the callee function executes the `RET` instruction, control is passed back to the caller function. In some cases, there is no return value (if the callee is declared of type `void`) and, in some cases, the caller function ignores the return value. Again, from our previous example, the return value is assigned to the variable `w` in `Watt`.

In particular, there are two actions that must be performed:

1. The return value (if there is one) is popped off the stack.
2. The arguments are popped off the stack.

The code after the `JSR` looks like the following:

```
JSR  Volta

LDR  R0, R6, #0   ; Load the return value
                  ; at the top of stack
STR  R0, R5, #0   ; w = Volta(w, 10);
ADD  R6, R6, #1   ; Pop return value

ADD  R6, R6, #2   ; Pop arguments
```

Once this code is done, the call is now complete and the caller function can resume its normal operation. Notice that prior to the return to the caller, the callee restores the environment of the caller. To the caller, it appears as if nothing has changed except that a new value (the return value) has been pushed onto the stack.

Caller Save/Callee Save

Before we complete our discussion of the implementation of functions, we need to cover a topic that we've so far swept under the rug. During the execution of a function, R0 through R3 can contain temporary values that are part of an ongoing computation. Registers R4 through R7 are reserved for other purposes: R4 is the pointer to the global data section, R5 is the frame pointer, R6 is the stack pointer, and R7 is used to hold return addresses. If we make a function call, based on the calling convention we've described R4 through R7 do not change or change in predetermined ways. But what happens to registers R0, R1, R2, and R3? In the general case, we'd like to make sure that the callee function does not overwrite them. To address this, calling conventions typically adopt one of two strategies: (1) The caller will save these registers by pushing them onto its activation record. This is called the *caller-save* convention. (We also discussed this in Chapter 9.) When control is returned to the caller, the caller will restore these registers by popping them off the stack. (2) Alternatively, the callee can save these registers by adding four fields in the bookkeeping area of its record. This is called the *callee-save* convention. When the callee is initiated, it will save R0 through R3 and R5 and R7 into the bookkeeping region and restore these registers prior to the return to the caller.

14.3.3 Tying It All Together

The code for the function call in `Watt` and the beginning and end of `Volta` is listed in Figure 14.8. The LC-3 code segments presented in the previous sections are all combined, showing the overall structure of the code. This code is more optimized than the previous individual code segments. We've combined the manipulation of the stack pointer R6 associated with pushing and popping the return value into single instructions.

To summarize, our LC-3 C calling convention involves a series of steps that are performed when a function calls another function. The caller function pushes the value of each parameter onto the stack and performs a Jump To Subroutine (`JSR`) to the callee. The callee allocates a space for the return value, saves some bookkeeping information about the caller, and then allocates space on the stack for its local variables. The callee then proceeds to carry out its task. When the task is complete, the callee writes the return value into the space reserved for it, pops and restores the bookkeeping information, and returns to the caller. The caller then pops the return value and the parameters it placed on the stack and resumes its execution.

You might be wondering why we would go through all these steps just to make a function call. That is, is all this code really required and couldn't the calling convention be made simpler? One of the characteristics of real calling conventions is that in the general case, any function should be able to call any other function. To enable this, the calling convention should be organized so that a caller does not need to know anything about a callee except its interface (that is, the type of value the callee returns and the types of values it expects as parameters). Likewise, a callee is written to be independent of the functions that call it. Because of this generality, the calling convention for C functions require the steps we have outlined here.

```
1   Watt:
2     ...
3     AND  R0, R0, #0   ; R0 <- 0
4     ADD  R0, R0, #10  ; R0 <- 10
5     ADD  R6, R6, #-1  ;
6     STR  R0, R6, #0   ; Push 10
7     LDR  R0, R5, #0   ; Load w
8     ADD  R6, R6, #-1  ;
9     STR  R0, R6, #0   ; Push w
10
11    JSR  Volta
12
13    LDR  R0, R6, #0   ; Load the return value at top of stack
14    STR  R0, R5, #0   ; w = Volta(w, 10);
15    ADD  R6, R6, #3   ; Pop return value, arguments
16    ...
17
18  Volta:
19    ADD R6, R6, #-2   ; Push return value
20    STR R7, R6, #0    ; Push return address
21    ADD R6, R6, #-1   ; Push R5 (Caller's frame pointer)
22    STR R5, R6, #0    ; We call this the dynamic link
23    ADD R5, R6, #-1   ; Set new base pointer
24    ADD R6, R6, #-2   ; Allocate memory for Volta's locals
25
26    ...               ; Volta performs its work
27
28    LDR  R0, R5, #0   ; Load local variable k
29    STR  R0, R5, #3   ; Write it in return value slot
30    ADD  R6, R5, #1   ; Pop local variables
31    LDR  R5, R6, #0   ; Pop the dynamic link
32    ADD  R6, R6, #1   ;
33    LDR  R7, R6, #0   ; Pop the return address
34    ADD  R6, R6, #1   ;
35    RET
```

Figure 14.8 The LC-3 code corresponding to a C function call and return

14.4 Problem Solving Using Functions

For functions to be useful to us, we must somehow integrate them into our programming problem-solving methodology. In this section we will demonstrate the use of functions through two example problems, with each example demonstrating a slightly different application of functions.

Conceptually, functions are a good point of division during the top-down design of an algorithm from a problem. As we decompose a problem, natural "components" will appear in the tasks that are to be performed by the algorithm. And these components are natural candidates for functions. Our first example involves converting text from lowercase into uppercase, and it presents an

example of a component function that is naturally apparent during the top-down design process.

Functions are also useful for encapsulating primitive operations that the program requires at various spots in the code. By creating such a function, we are in a sense extending the set of operations of the programming language, tailoring them to the specific problem at hand. In the case of the second problem, which determines Pythagorean Triples, we will develop a primitive function to calculate x^2 to assist with the calculation.

14.4.1 Problem 1: Case Conversion

In this section, we go through the development of a program that reads input from the keyboard and echos it back to the screen. We have already seen an example of a program that does just this in Chapter 13 (see Figure 13.8). However, this time, we throw in a slight twist: We want the program to convert lowercase characters into uppercase before echoing them onto the screen.

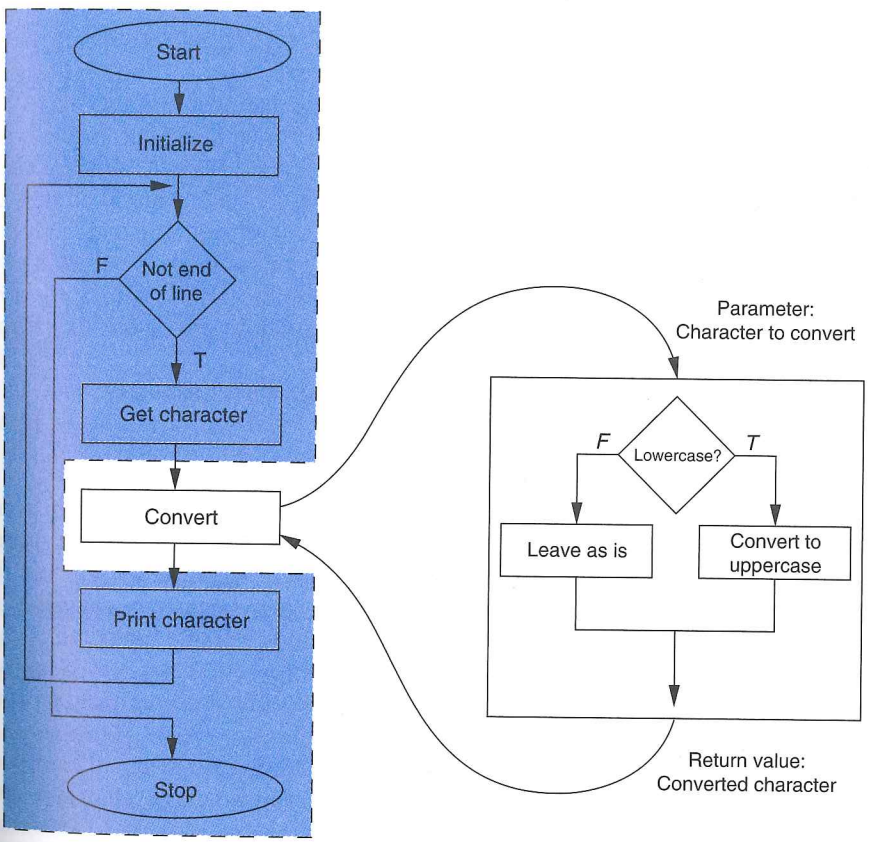

Figure 14.9 The decomposition into smaller subtasks of a program that converts input characters into uppercase

```
1  #include <stdio.h>
2
3  /* Function declaration */ char ToUpper(char inchar);
4                                                              */
5  /* Function main:                                          */
6  /* Prompt for a line of text, Read one character,          */
7  /* convert to uppercase, print it out, then get another    */
8  int main()
9  {
10   char echo = 'A';              /* Initialize input character */
11   char upcase;                      /* Converted character    */
12
13   while (echo != '\n') {
14     scanf("%c", &echo);
15     upcase = ToUpper(echo);
16     printf("%c", upcase);
17   }
18 }
19                                                             */
20 /* Function ToUpper:                                       */
21 /* If the parameter is lower case return                   */
22 /* its uppercase ASCII value
23 char ToUpper(char inchar)
24 {
25   char outchar;
26
27   if ('a' <= inchar && inchar <= 'z')
28     outchar = inchar - ('a' - 'A');
29   else
30     outchar = inchar;
31
32   return outchar;
33 }
```

Figure 14.10 A program with a function to convert lowercase letters to uppercase

Our approach to solving this problem is to use the echo program from Figure 13.8 as a starting point. The previous code used a `while` loop to read an input character from the keyboard and then print it to the output device. To this basic structure, we want to add a component that checks if a character is lowercase and converts it to uppercase if it is. There is a single input and a single output. We could add code to perform this directly into the `while` loop, but given the self-contained nature of this component, we will create a function to do this job.

The conversion function is called after each character is scanned from the keyboard and before it is displayed to the screen. The function requires a single character as a parameter and returns either the same character (for cases in which the character is already uppercase or is not a character of the alphabet) or it will return an uppercase version of the character. Figure 14.9 shows

the flow of this program. The flowchart of the original echo program is shaded. To this original flowchart, we are adding a component function to perform the conversion.

Figure 14.10 shows the complete C program. It takes input from the keyboard, converts each input character into uppercase, and prints out the result. When the input character is the new line character, the program terminates. The conversion process from lowercase to uppercase is done by the function `ToUpper`. Notice the use of ASCII literals in the function body to perform the actual conversion. Keep in mind that a character in single quotes (e.g., `'A'`) is evaluated as the ASCII value of that character. The expression `'a' - 'A'` is therefore the ASCII value of the character `a` minus the ASCII of `A`.

14.4.2 Problem 2: Pythagorean Triples

Now we'll attempt a programming problem involving calculating all Pythagorean Triples less than a particular input value. A Pythagorean Triple is a set of three **integer** values a, b, and c that satisfy the property $c^2 = a^2 + b^2$. In other words, a and b are the lengths of the sides of a right triangle where c is the hypotenuse. For example, 3, 4, and 5 is a Pythagorean Triple. The problem here is to calculate all Triples a, b, and c where all are less than a limit provided by the user.

For this problem, we will attempt to find all Triples by brute force. That is, if the limit indicated by the user is `max`, we will check all combinations of three integers less than `max` to see if they satisfy the Triple property. In order to check all combinations, we will want to vary each `sideA`, `sideB`, and `sideC` from 1 to `max`. This implies the use of counter-controlled loops. More exactly, we will want to use a `for` loop to vary `sideC`, another to vary `sideB`, and another to vary `sideA`, each *nested* within the other. At the core of these loops, we will check to see if the property holds for the three values, and if so, we'll print them out.

Now, in performing the Triple check, we will need to evaluate the following expression.

```
(sideC * sideC == (sideA * sideA + sideB * sideB))
```

Because the square operation is a primitive operation for this problem—meaning it is required in several spots—we will encapsulate it into a function `Squared` that returns the square of its integer parameter. The preceding expression will be rewritten as follows. Notice that this code gives a clearer indication of what is being calculated.

```
(Squared(sideC) == Squared(sideA) + Squared(sideB))
```

The C program for this is provided in Figure 14.11. There are better ways to calculate Triples than with a brute-force technique of checking all combinations (Can you modify the code to run more efficiently?); the brute-force technique suits our purposes of demonstrating the use of functions.

```
#include <stdio.h>

int Squared(int x);

int main()
{
  int sideA;
  int sideB;
  int sideC;
  int maxC;

  printf("Enter the maximum length of hypotenuse: ");
  scanf("%d", &maxC);

  for (sideC = 1; sideC <= maxC; sideC++) {
    for (sideB = 1; sideB <= maxC; sideB++) {
      for (sideA = 1; sideA <= maxC; sideA++) {
        if (Squared(sideC) == Squared(sideA) + Squared(sideB))
          printf("%d %d %d\n", sideA, sideB, sideC);
      }
    }
  }

/* Calculate the square of a number */
int Squared(int x)
{
  return x * x;
}
```

Figure 14.11 A C program that calculates Pythagorean Triples

14.5 Summary

In this chapter, we introduced the concept of functions in C. The general notion of subprograms such as functions have been part of programming languages since the earliest languages. Functions are useful because they allow us to create new primitive building blocks that might be useful for a particular programming task (or for a variety of tasks). In a sense, they allow us to extend the native operations and constructs supported by the language.

The key notions that you should take away from this chapter are:

- **Syntax of functions in C.** To use a function in C, we must declare the function using a function declaration (which we typically do at the beginning of our code) that indicates the function's name, the type of value the function returns, and the types and order of values the function expects as inputs. A function's definition contains the actual code for the function. A function is invoked when a call to it is executed. A function call contains arguments—values that are to be passed to the function as parameters.

- **Implementation of C functions at the lower level.** Part of the complexity associated with implementing functions is that in C, a function can be called from any other function in the source file (and even from functions in other object files). To assist in dealing with this, we adopt a general calling convention for calling one function from another. To assist with the fact that some functions might even call themselves, we base this calling convention on the run-time stack. The calling convention involves the caller passing the value of its arguments by pushing them onto the stack, then calling the callee. The arguments written by the caller become the parameters of the callee's activation record. The callee does its task and then pops its activation record off the stack, leaving behind its return value for the caller.

- **Using functions when programming.** It is conceivable to write all your programs without ever using functions, the result would be that your code would be hard to read, maintain, and extend and would probably be buggier than if your code used functions. Functions enable abstraction: we can write a function to perform a particular task, debug it, test it, and then use it within the program whereever it is needed.

Exercises

14.1 What is the significance of the function `main`? Why must all programs contain this function?

14.2 Refer to the structure of an activation record for these questions.

a. What is the purpose of the dynamic link?
b. What is the purpose of the return address?
c. What is the purpose of the return value?

14.3 Refer to the C syntax of functions for these questions.

a. What is a function declaration? What is its purpose?
b. What is a function prototype?
c. What is a function definition?
d. What are arguments?
e. What are parameters?

14.4 For each of the following items, identify whether the caller function or the callee function performs the action.

a. Writing the parameters into the activation record.
b. Writing the return value.
c. Writing the dynamic link.
d. Modifying the value in R5 to point within the callee function's activation record.

14.5 What is the output of the following program? Explain.

```
void MyFunc(int z);

int main()
{
  int z = 2;

  MyFunc(z);
  MyFunc(z);
}

void MyFunc(int z)
{
  printf("%d ", z);
  z++;
}
```

14.6 What is the output of the following program?

```
#include <stdio.h>

int Multiply(int d, int b);

int d = 3;

int main()
{
  int a, b, c;
  int e = 4;

  a = 1;
  b = 2;

  c = Multiply(a, b);
  printf("%d %d %d %d %d\n", a, b, c, d, e);
}

int Multiply(int d, int b)
{
  int a;
  a = 2;
  b = 3;

  return (a * b);
}
```

14.7 Following is the code for a C function named `Bump`.

```
int Bump(int x)
{
   int a;

   a = x + 1;

   return a;
}
```

a. Draw the activation record for `Bump`.

b. Write one of the following in each entry of the activation record to indicate what is stored there.
   (1) Local variable
   (2) Argument
   (3) Address of an instruction
   (4) Address of data
   (5) Other

c. Some of the entries in the activation record for `Bump` are written by the function that calls `Bump`; some are written by `Bump` itself. Identify the entries written by `Bump`.

14.8 What is the output of the following code? Explain why the function `Swap` behaves the way it does.

```
int main()
{
   int x = 1;
   int y = 2;

   Swap(x, y);
   printf("x = %d   y = %d\n", x, y);
}

void Swap(int y, int x)
{
  int temp

  temp = x;
  x = y;
  y = temp;
}
```

14.9 Are the parameters to a function placed on the stack before or after the JSR to that function? Why?

14.10 A C program containing the function `food` has been compiled into LC-3 assembly language. The partial translation of the function into LC-3 is:

```
food:
  ADD R6, R6, #-2   ;
  STR R7, R6, #0    ;
  ADD R6, R6, #-1   ;
  STR R5, R6, #0    ;
  ADD R5, R6, #-1   ;
  ADD R6, R6, #-4   ;
  ...
```

a. How many local variables does this function have?
b. Say this function takes two integer parameters `x` and `y`. Generate the code to evaluate the expression `x + y`.

14.11 Following is the code for a C function named `Unit`.

```
int main()
{
  int a = 1;
  int b = 2;

  a = Init(a);
  b = Unit(b);

  printf("a = %d   b = %d\n", a, b);
}

int Init(int x)
{
   int y = 2;

   return y + x;
}

int Unit(int x)
{
   int z;

   return z + x;
}
```

a. What is the output of this program?
b. What determines the value of local variable `z` when function `Unit` starts execution?

14.12 Modify the example in Figure 14.10 to also convert each character to lowercase. The new program should print out both the lower- and uppercase versions of each input character.

14.13 Write a function to print out an integer value in base 4 (using only the digits 0, 1, 2, 3). Use this function to write a program that reads two integers from the keyboard and displays both numbers and their sum in base 4 on the screen.

14.14 Write a function that returns a 1 if the first integer input parameter is evenly divisible by the second. Using this function, write a program to find the smallest number that is evenly divisible by all integers less than 10.

14.15 The following C program is compiled into LC-3 machine language and loaded into address x3000 before execution. Not counting the JSRs to library routines for I/O, the object code contains three JSRs (one to function `f`, one to `g`, and one to `h`). Suppose the addresses of the three JSR instructions are x3102, x3301, and x3304. And suppose the user provides `4 5 6` as input values. Draw a picture of the run-time stack, providing the contents of locations, if possible, when the program is about to return from function `f`. Assume the base of the run-time stack is location xEFFF.

```
#include <stdio.h>

int f(int x, int y, int z);
int g(int arg);
int h(int arg1, int arg2);

int main()
{
   int a, b, c;

   printf("Type three numbers: ");
   scanf("%d %d %d", &a, &b, &c);
   printf("%d", f(a, b, c));
}

int f(int x, int y, int z)
{
   int x1;

   x1 = g(x);
   return h(y, z) * x1;
}

int g(int arg)
{
   return arg * arg;
}

int h(int arg1, int arg2)
{
   return arg1 / arg2;
}
```

14.16 Referring once again to the machine-busy example from previous chapters, remember that we represent the busyness of a set of 16 machines with a bit pattern. Recall that a 0 in a particular bit position indicates the corresponding machine is busy and a 1 in that position indicates that machine is idle.

a. Write a function to count the number of busy machines for a given busyness pattern. The input to this function will be a bit pattern (which can be represented by an integer variable), and the output will be an integer corresponding to the number of busy machines.

b. Write a function to take two busyness patterns and determine which machines have changed state, that is, gone from busy to idle, or idle to busy. The output of this function is simply another bit pattern with a 1 in each position corresponding to a machine that has changed its state.

c. Write a program that reads a sequence of 10 busyness patterns from the keyboard and determines the average number of busy machines and the average number of machines that change state from one pattern to the next. The user signals the end of busyness patterns by entering a pattern of all 1s (all machines idle). Use the functions you developed for parts 1 and 2 to write your program.

14.17 *a.* Write a C function that mimics the behavior of a 4-to-1 multiplexor. See Figure 3.13 for a description of a 4-to-1 MUX.

b. Write a C function that mimics the behavior of the LC-3 ALU.

14.18 Notice that on a telephone keypad, the keys labeled 2, 3, 4, . . ., 9 also have letters associated with them. For example, the key labeled 2 corresponds to the letters *A*, *B*, and *C*. Write a program that will map a seven-digit telephone number into all possible character sequences that the phone number can represent. For this program, use a function that performs the mapping between digits and characters. The digits 1 and 0 map to nothing.

14.19 The following C program uses a combination of global variables and local variables with different scope. What is the output?

```
#include <stdio.h>
int t = 1;              /* Global variable */
int sub1(int fluff);
int main ()
{
    int t = 2;
    int z;
    z = t;
    z = z + 1;
    printf("A: The variable z equals %d\n", z);

    {
        z = t;
        t = 3;

        {
            int t = 4;
            z = t;
            z = z + 1;
            printf("B: The variable z equals %d\n", z);
        }

        z = sub1(z);
        z = z + 1;
        printf("C: The variable z equals %d\n", z);
    }
    z = t;
    z = z + 1;
    printf("D: The variable z equals %d\n", z);
}

int sub1(int fluff)
{
    int i;
    i = t;
    return (fluff + i);
}
```